Q&A

Principi Object Oriented

▼ Concetto di ereditarietà

L’ereditarietà è un concetto di programmazione Object Oriented attraverso il quale le classi possono essere organizzate in una gerarchia. Gli oggetti delle sottoclassi devono essere in grado di comportarsi allo stesso modo della superclasse e hanno la possibilità di avere caratteristiche aggiuntive rispetto alla superclasse o modificarne alcune funzionalità, a patto che queste modifiche non siano visibili dall’esterno.

L’ereditarietà può essere:

- Di interfaccia o di estensione: garantisce la compatibilità tra tipi e consente il polimorfismo per inclusione
- Di realizzazione: si riutilizza il codice delle superclassi. Non è ammessa in Java e .NET, ma in C++ sì. Un’alternativa al concetto di ereditarietà di realizzazione è la Composizione e Delega in cui una classe contiene l’altra, avendo accesso indiretto alle sue operazioni pubbliche. Le interfacce delle classi restano però indipendenti.

▼ Polimorfismo

Per polimorfismo si intende la possibilità di un oggetto di apparire in forme diverse a seconda del contesto, e di oggetti diversi di apparire sotto la stessa forma. La classificazione di Cardelli-Wegner del polimorfismo individua due categorie, a loro volta divise in due sottocategorie:

- **Polimorfismo Universale** - gli elementi assumono infinite forme
  - **Per inclusione**
    Utilizzato nella programmazione a oggetti, utilizza:
    - **Overriding** (Definizione di metodi astratti o ridefinizione di metodi concreti)
    - **Binding dinamico**
    - **Virtual Method Table** (Struttura utilizzata per il binding dinamico), posseduta da ogni classe e contiene tutti i puntatori ai metodi della classe

    - **Parametrico**
      Utilizzato nella programmazione generica rispetto ai tipi. Si definisce una classe in cui una o più variabili sono un parametro della classe stessa. Ogni istanza di una classe generica è indipendente e non ha alcun rapporto di ereditarietà con le altre
- **Polimorfismo Ad Hoc** - gli elementi assumono un numero finito di forme
    - **Overloading**
      Consente la ridefinizione di metodi per ogni insieme di argomenti accettati
    - **Coercion**
      Conversione implicita del tipo in una variabile

▼ Ereditarietà multipla

Si verifica quando, in una gerarchia di classi, una classe deriva da più di una superclasse. In Java e C# non è consentita a causa della generazione di ambiguità e la complessa gestione della gerarchia che ne deriverebbe. In C++ è utilizzata.

Uno dei classici problemi dell'ereditarietà multipla è che, se una classe con due sottoclassi distinte definisce un metodo che viene ridefinito da entrambe, una classe che eredita da entrambe le sottoclassi non saprebbe qual è il metodo corretto.

Java e C# adottano l'ereditarietà multipla delle interfacce, in quanto all'interno di queste ultime non si trova la definizione di metodi, bensì unicamente la loro interfaccia.

Un'altra soluzione all'ereditarietà multipla è il modello Composizione e Delega, in cui si sceglie una superclasse rilevante da cui ereditano le sottoclassi e le superclassi rimanenti diventano ruoli e sono connesse tramite composizione.

.NET e C#

▼ Differenze tra Tecnologia COM e Framework .NET

COM è un sistema platform-indipendent, distribuito e object oriented per la creazione di componenti software binari. E' uno standard e non un linguaggio orientato agli oggetti, specifica un modello a oggetti e requisiti di programmazione per far interagire oggetti COM con altri. L'ereditarietà avviene solo tramite composizione e delega.

COM utilizza la tecnica del Reference Counting per assicurare che gli oggetti esistano solo per il tempo in cui esistono i client che li utilizzano e hanno accesso alle loro interfacce. Un oggetto COM è responsabile della liberazione della propria memoria. La posizione di ciascun componente è memorizzata nel registro windows, il che porta a complicanze durante il deployment quando ci sono programmi che funzionano con diverse versioni di uno stesso componente COM.

.NET, invece, è costituito da un ambiente di esecuzione e una libreria di classi. E' completamente indipendente da COM, ma integrato con esso. Definisce funzionalità comuni per tutti i linguaggi del framework, così che i componenti possano essere scritti in linguaggi differenti. E' un ambiente Object Oriented che fornisce le seguenti funzionalità:

- Garbage Collector

  Non utilizza il Reference Counting. Questo porta a più velocità di allocazione, ma la distruzione degli oggetti non è più deterministica. Sono consentiti i riferimenti circolari.

- Type Checker
- Gestione delle Eccezioni
- I/O su file

.NET è un'implementazione di CLI - Common Language Infrastructure, in cui applicazioni scritte in linguaggi diversi vengono eseguite in ambienti diversi senza dover riscrivere l'applicazione. L'ambiente a tempo di esecuzione si compone di:

- **MSIL - Microsoft Intermediate Language**
- **CLR - Common Language Runtime**
  E' un ambiente di esecuzione runtime per le applicazioni .NET. Il codice eseguito nel CLR è detto codice gestito. Offre una vasta gamma di funzionalità, sia specifiche che già esistenti e mediate da CLR.
- **CTS - Common Type System**
  Definisce i tipi supportati dal framework e un modello di programmazione unificato per linguaggi a oggetti, procedurali o funzionali. .NET è fortemente tipizzato e fornisce ereditarietà singola di estensione e multipla di interfaccia. In .NET tutto è un oggetto, con classe radice System.Object. Esistono sia tipi riferimento (nell'heap, identificati da indirizzi di memoria) che tipi valore (nello stack, identificati da sequenze di byte). Si può effettuare la conversione

esplicita da tipo valore a tipo riferimento tramite up cast implicito a System.Object (`double d = 3.33; Object o = d;`) oppure tra tipo riferimento a tipo valore tramite down cast esplicito (`double d2 = (double) obj`)

- **CLS - Common Language Specification** (In CTS, regole per i linguaggi per poter interoperare in .NET)

Il codice in .NET può essere interpretato, nativo o codice IL (CLI assembly, ovvero .exe o .dll). L'assembly contiente sia codice che metadati. Il codice IL viene convertito dal compilatore JIT (Just in Time) in codice nativo.

I metadati sono composti dal manifest, che descrive l'assembly specificandone identità, lista dei file, riferimenti ad altri assembly, permessi, …, e definizione dei tipi di dati contenuti nell'assembly, che vengono utilizzati da compilatore, ambienti RAD e tool di analisi. L'assembly può essere **privato** se utilizzato solo da un'applicazione specifica, **condiviso** se utilizzato da più applicazioni (si trova nella Global Assembly Cache (GAC)) o **scaricato da URL**.

.NET permette il deployment semplificato: applicazioni e componenti possono essere condivisi o privati e diverse versioni dello stesso componente possono coesistere.

▼ Garbage Collector

Per utilizzare un oggetto in un linguaggio a oggetti, bisogna:

1. allocare la memoria
2. inizializzarla
   A ogni variabile va assegnato un valore prima di utilizzarla. La verifica viene effettuata dal compilatore.
3. utilizzare l'oggetto
4. effettuare una pulizia dello stato dell'oggetto utilizzando un distruttore unico, non ereditabile, senza overload e invocato automaticamente
5. liberare la memoria

La presenza di un garbage collector porta numerosi vantaggi, tra cui la risoluzione di problemi come dangling pointer, doppia deallocazione o memory leak. Tuttavia, aumenta la richiesta di risorse di calcolo e il rilascio della memoria non è più deterministico e difficile da prevedere.

Esistono diverse strategie per l'implementazione di un Garbage Collector:

- **Tracing**
  Si determinano gli oggetti raggiungibili e si eliminano gli oggetti garbage. Un oggetto è detto raggiungibile se è un oggetto radice, quindi se è creato all'avvio del programma o da una subroutine, o se è referenziato da un oggetto raggiungibile. Ciascuna radice è un puntatore a un oggetto di tipo riferimento sicuramente attivo (es. variabili globali, campi statici di tipo riferimento, variabili locali o argomenti attuali di tipo riferimento sugli stack dei vari thread, registri CPU).
  Dato che la definizione di Garbage Collector tramite raggiungibilità non è ottimale, in quanto un programma potrebbe utilizzare un oggetto molto prima che diventi irraggiungibile, si distingue tra: **Garbage Sintattico** (l'oggetto non può essere raggiunto) e **Garbage Semantico** (l'oggetto non viene usato dal programma)
- **Escape Analysis**
  Si spostano gli oggetti dall'heap allo stack. Viene effettuata un'analisi a tempo di compilazione per capire se un oggetto è raggiungibile al di fuori della subroutine in cui è stato allocato. Riduce molto il lavoro del Garbage Collector
- **Reference Counting**
  Ogni oggetto conserva un contatore che indica il numero di riferimenti a esso. Quando il contatore è zero può essere eliminato. E' una soluzione che porta innumerevoli svantaggi, tra cui: impossibilità di effettuare riferimenti ciclici, occupazione di memoria, riduzione della velocità delle operazioni, necessità che le operazioni siano atomiche, assenza di comportamento real time.

In **CLR**, viene riservato uno spazio di memoria contiguo (**Managed Heap**) e si memorizza in un puntatore il punto di partenza dell'area.

Quando viene creato un nuovo oggetto, viene calcolata la sua dimensione e si aggiungono al suo interno un puntatore alla tabella dei metodi e un field `SyncBlockIndex`. Successivamente, si controlla che ci sia spazio sufficiente ad allocare tale oggetto. Se non c'è entra in azione il GC o viene lanciata una `OutOfMemoryException`. Successivamente, si memorizza il puntatore all'oggetto: `thisObjPtr=nextObjPtr` e si calcola il nuovo `nextObjPtr`. Infine viene invocato il costruttore e l'oggetto viene restituito.
Il Garbage Collector controlla se ci sono oggetti non più utilizzati: ogni applicazione ha un insieme di radici (puntatori con l'indirizzo dell'oggetto o nulli). Gli oggetti sono vivi se raggiungibili direttamente o indirettamente dalle radici,

altrimenti sono oggetti garbage.
Inizialmente, tutti gli oggetti sono garbage. Il GC analizza tutti gli oggetti e segna quelli raggiungibili come vivi. Quando libera la memoria, la ricompatta e aggiorna il `nextObjPtr`. I riferimenti a risorse non gestite vengono gestiti dal programmatore (es. file, connessioni, …)

▼ Passaggio dei parametri in C#

In C# ci sono tre tipi di argomenti che possono essere passati ai metodi:

- In (`default` in C#)
  - L'argomento deve essere inizializzato
  - L'argomento viene passato per valore
  - Eventuali modifiche del valore dell'argomento non hanno effetto sul chiamante
  - Se l'argomento è di tipo riferimento viene passata una copia del riferimento all'oggetto. Eventuali modifiche dell'oggetto referenziato hanno effetto. Eventuali modifiche del riferimento vengono effettuate sulla copia e non hanno effetto sul riferimento originale
- In/Out (`ref` in C#)
  - L'argomento deve essere inizializzato
  - L'argomento viene passato per riferimento
  - Eventuali modifiche sul valore dell'argomento hanno effetto sul chiamante
  - Se l'argomento è di tipo riferimento viene passato l'indirizzo del riferimento all'oggetto. Eventuali modifiche dell'oggetto referenziato hanno effetto. Eventuali modifiche del riferimento agiscono direttamente sul riferimento originale
- Out(`out` in C#)
  - L'argomento può non essere inizializzato
  - L'argomento viene passato per riferimento
  - Le modifiche sul valore hanno effetto sul chiamante
  - Inizializzazione obbligatoria nel metodo

- Viene passato l’indirizzo dell’oggetto o del riferimento all’oggetto come nel caso In/Out

E’ buona norma utilizzare prevalentemente il passaggio standard per valore e utilizzare il passaggio per riferimento solo se bisogna restituire più di un valore al chiamante, ci sono uno o più valori da utilizzare e modificare nel metodo. Se l’oggetto è stato inizializzato è bene usare `ref`, altrimenti `out`.

▼ Delegati ed Eventi

I delegati sono oggetti che possono contenere il riferimento a un metodo tramite cui può essere invocato (come i puntatori a funzione, ma object-oriented e più potenti). Sono utili per funzioni di callback come elaborazione asincrona, cooperativa e gestione degli eventi (chi è interessato a un certo evento si registra presso il generatore dell’evento, specificando il metodo che lo gestirà).

```
delegate int Azione(int param);

Azione azione;
azione = new Azione(nomeMetodoStatico);
azione = new Azione(obj.nomeMetodo);

int k = azione(10);
```

Un’istanza di un delegato contiene uno o più metodi, ciascuno riferito a un’entità richiamabile. Per i metodi statici, l’istanza invocabile consiste di un metodo. Un delegato applica solo una firma del metodo e non conosce la classe dell’oggetto a cui fa riferimento. Ciò rende i delegati utili per chiamate anonime. E’ possibile assegnare a un delegato una lista di metodi che verranno invocati in sequenza e in modo sincrono.

```
Azione azione = Fun1;
azione += Fun2;
azione -= Fun2;
```

A ogni metodo vengono passati i parametri forniti al delegato. Per parametri di tipo riferimento, ogni invocazione del metodo avviene con un riferimento alla stessa variabile e le modifiche da parte di un metodo a quella variabile saranno visibili ai metodi successivi. Se la chiamata ha un valore di ritorno o parametri out, questi sono dati dall’ultimo metodo della lista.

In C#, la dichiarazione di un nuovo tipo di delegato definisce automaticamente una nuova classe derivata dalla classe `System.MulticastDelegate`. Pertanto, sulle istanze di `Azione` è possibile invocare metodi definiti a livello di classi di sistema.

Un esempio di utilizzo di delegati è il concetto di Boss-Worker, in cui un Worker effettua un'attività e il Boss controlla l'attività dei suoi Worker. Il Worker notificherà il Boss all'inizio, durante e alla fine del lavoro.

Un delegato è un'entità type-safe che si pone tra 1 caller e 0+ call target e agisce come un'interfaccia con un solo metodo.

Utilizzare campi pubblici per la registrazione non è sicuro, in quanto i client possono sovrascrivere altri client registrati precedentemente o invocare i chiamati. Una soluzione sarebbe quella di fornire metodi di registrazione pubblica abbinati al campo del delegato, ma è una soluzione pesante da implementare.

Eventi

L'utilizzo di eventi automatizza il supporto per la registrazione pubblica e l'implementazione privata.

E' possibile definire gestori per registrarsi a eventi, con il vantaggio di avere più controllo, una sintassi alternativa che supporta gestori definiti dall'utente, la possibilità di registrazione condizionale o personalizzata. L'accesso lato client resta uguale. Tuttavia, c'è bisogno di più spazio per i clienti registrati.

Esistono tre entità relative agli eventi:

- Event Sender
- Event Receiver
- Event Handler

Quando si verifica un evento, il sender invia una notifica a tutti i receiver. Il sender non conosce né i receiver né gli handler, ma è collegato a loro tramite il meccanismo dei delegati.

Un evento incapsula un delegato, perciò bisogna prima dichiarare un tipo di delegato per dichiarare un evento. I delegati in .NET hanno due parametri: la sorgente dell'evento e i dati dell'evento. Se l'evento non genera dati, basta utilizzare il delegato dell'evento System.EventHandler.

```
public delegate void EventHandler(object sender, EventArgs e);
```

Si utilizza `System.EventArgs` solo se un evento non deve passare informazioni aggiuntive ai propri gestori. In caso contrario, si utilizza una classe che estende `EventArgs` e il delegato `EventHandler<TEventArgs>`.

```
public event EventHandler Changed;
```

La keyword event limita la visibilità e le possibilità di utilizzo del delegato. Dopo la dichiarazione, l'evento è un delegato che può essere null se non ha client registrati o essere associato a uno o più metodi da invocare.

Rispetto ai delegati, l'invocazione di un evento può avvenire solo all'interno della classe in cui è stato dichiarato (anche se è public). Fuori dalla classe, è un delegato con accessi molto limitati. Si può solo agganciarsi a un nuovo evento (aggiungere un nuovo delegato all'evento) o sganciarsi da esso.

Per ricevere le notifiche di un evento, il client deve: definire il metodo event handler che verrà invocato e creare un delegato dello stesso tipo dell'evento, farlo riferire al metodo e aggiungerlo alla lista dei delegati per l'evento.

```
void ListChanged(object sender, EventArgs e) {...}

List.Changed += new EventHandler(ListChanged)
//Oppure
List.Changed += ListChanged;
```

Gli eventi forniscono agli oggetti un modo utile per segnalare modifiche allo stato ai clienti. Sono un componente fondamentale per la creazione di classi che possono essere riutilizzate in un gran numero di programmi diversi.

▼ Metadati e Introspezione

La metaprogrammazione viene utilizzata per programmare un sistema in modo che abbia accesso a informazioni sul sistema stesso e possa manipolarle. Viene realizzata tramite il meccanismo della **Reflection**, implementata in C# tramite la classe `System.Reflection`. Questo meccanismo sfrutta i metadati. COM e CORBA utilizzano l'IDL (Interface Definition Language) per fornire metadati. Se un componente dispone di tutte le informazioni per autodescriversi, le sue interfacce possono essere esplorate dinamicamente.

I metadati sono generati dalla definizione del tipo e memorizzati assieme alla sua definizione per essere disponibili a runtime.

La riflessione viene utilizzata per:

- Esaminare i dettagli di un assembly
- Istanziare oggetti e chiamare metodi scoperti a runtime
- Creare, compilare ed eseguire assembly

La chiave della riflessione in C# è la classe `System.Type`. Tutti gli oggetti sono istanze di tipi e, al tempo stesso, istanze di `System.Type`. Il tipo di un oggetto può essere scoperto tramite il metodo `getType` ed esiste un solo oggetto `Type` per ogni tipo definito.

Ad esempio, è possibile enumerare tutti i tipi di un assembly attraverso i seguenti comandi:

- `Assembly.Load` per caricare un assembly .NET. Restituisce un'istanza Assembly
- `Assembly.GetModules` per ottenere un array di Module
- Per ogni Module, invocare `Module.getTypes` per ottenere un array di `Type`

E' possibile creare istanze di tipi e accedere ai membri in modo very late bound:

- `Activator.CreateInstance(type, …)` invoca il costruttore specificato
- `methodInfo.Invoke(…)` invoca i metodi
- `propertyInfo.GetValue(…) propertyInfo.SetValue(…)` invocano accessori e modificatori di proprietà
- I membri pubblici sono sempre accessibili
- I membri non pubblici sono accessibili solo se si hanno i permessi

Per aggiungere informazioni ai metadati è possibile utilizzare attributi personalizzati, cioè classi visibili via riflessione che derivano da `System.Attribute` e possono contenere proprietà e metodi. Occorre:

- Dichiarare la classe dell'attributo
- Dichiarare i costruttori
- Dichiarare le proprietà
- Opzionalmente, applicare `AttributeUsageAttribute`, che specifica alcune caratteristiche della classe, ovvero a quali elementi l'attributo è applicabile, quando può essere ereditato, quando possono esistere molteplici istanze di un attributo

La classe `System.Reflection.Emit` consente di scrivere il codice IL necessario per creare e compilare un assembly che può essere chiamato direttamente dal programma che lo ha creato.

Architettura di un Sistema Software

▼ Struttura a livelli di un’applicazione

In un’applicazione troviamo diversi livelli, che vanno dal livello più vicino all’utente a quello più vicino alla base di dati.

- **Presentation Manager**
  - Gestisce l’interazione con l’utente
    - Interfacce grafiche
    - Interfacce a caratteri
- **Presentation Logic**
  - Gestisce l’interazione a livello logico
    - Consistenza dei dati
    - Accettazione dei dati
    - Validazione dei dati
    - Gestione dei messaggi di errore
- **Application Logic**
  - Contiene la logica dell’applicazione e il controllo dei componenti
- **Data Logic**
  - Gestisce la persistenza a livello logico
    - Consistenza dei dati
    - Apertura e chiusura dei file
    - Istruzioni SQL e transazioni
    - Gestione degli errori
- **Data Manager**
  - Gestisce la persistenza tramite il file system o RDBMS

Tra due livelli dell’applicazione ci può essere un **middleware**, cioè un software per la comunicazione tra processi dotato di API che isolano il codice da formati e protocolli di comunicazione dei livelli sottostanti. In particolare:

- **Presentation Middleware,** tra Presentation Manager e Presentation Logic, è un software di emulazione di terminali alfanumerici o una combinazione di web browser, web server e protocolli HTTP.
- **Application Middleware**, all’interno dell’Application Logic, si occupa di gestire la comunicazione tra processo dell’applicazione e interconnettere tipi di middleware diversi (es. P2P, RPC, DTM)
- **Database Middleware**, tra Data Logic e Data Manger, è un software proprietario che trasferisce sulla rete le richieste SQL dall’applicazione al DBMS e i dati nel verso opposto

Design Principles

▼ Bad Design

La qualità del design è fondamentale per ottenere un software affidabile, efficiente, manutenibile ed economico. Esistono quattro principi di design cattivo che peggiorano la qualità del software.

- Rigidità del software - il programma è difficile da modificare perché ogni piccola modifica richiederebbe altre modifiche a cascata sui moduli dipendenti tra loro.
- Fragilità del software - ogni modifica in una parte del sistema rende inutilizzabili altre parti del progetto a causa dell’insorgere di malfunzionamenti. Questo accade se i moduli sono fortemente dipendenti tra di loro
- Immobilità del software - il software o i componenti sono inutilizzabili in altre parti del sistema perché fortemente dipendenti da altri moduli
- Viscosità del software - cambiamenti che non preservano il design dell’applicazione e non seguono le best practice ma sono più hack. Distinguiamo tra:
    - Viscosità di design - si implementano hack invece che metodi che mantengono il design
    - Viscosità dell’ambiente - ambiente di sviluppo lento e inefficace

- Inoltre, un software è di scarsa qualità soprattutto se non rispetta i requisiti

Le cause di una progettazione errata sono:

- La scarsa conoscenza dei principi di design
- Il continuo cambiamento delle tecnologie
- Limiti di tempo e risorse
- La putrefazione del software è un processo lento che può durare anni
- Cambiamento continuo dei requisiti
- Dipendenze non pianificate

Per questo motivo è bene gestire le dipendenze tra i moduli dell'applicazione per evitare il degradamento dell'architettura.

▼ Principi di Design - Principio Zero

Il principio zero afferma che non si devono introdurre concetti che non siano strettamente necessari. Si deve puntare sempre alla semplicità: tra due soluzioni diverse bisogna optare sempre per quella che introduce meno ipotesi e concetti.

Vale il principio del **divide et impera**: dividere sempre un problema complesso in sottoparti semplici e facilmente risolvibili, minimizzando il grado di accoppiamento del sistema.

▼ Principio di Singola Responsabilità

Ogni elemento deve avere una singola responsabilità, che deve essere interamente incapsulata in esso. Tutti i servizi offerti dall'elemento dovrebbero essere allineati a tale responsabilità.

Il principio prevede di sviluppare classi o moduli indipendenti tra loro e semplici.

es. Stampante che è anche Fax → Classe Stampante e classe Fax

▼ Principio di Inversione delle Dipendenze

I moduli di alto livello non dovrebbero dipendere da quelli di basso livello, ma entrambi dovrebbero dipendere da astrazioni.

I moduli di basso livello sono più soggetti a modifiche. Se hanno delle dipendenze, i cambiamenti si propagano in tutti i moduli dipendenti da essi. Ciò porta a rigidità, fragilità e immobilità del software.

Questo principio funziona perché le astrazioni hanno poco codice e sono poco soggette a modifiche e cambiamenti. I cambiamenti dei moduli non astratti, così,

sono sicuri perché nessuno dipende da questi.

- I cambiamenti non possono più propagarsi → Design for change
- I moduli sono maggiormente riutilizzabili → Design for reuse

▼ Principio di Segregazione delle Interfacce

Un cliente non deve dipendere da moduli che non usa. Questo principio porta ad avere interfacce piccole e più specifiche, anziché interfacce enormi con più funzioni (fat interface), in quanto le fat interface accoppiano involontariamente i vari client e rendono più difficile manutenere il sistema. Rispettando questo principio ogni modifica di una funzione risulta più semplice, in quanto è autocontenuta.

▼ Principio Aperto/Chiuso

Il principio afferma che le entità del sistema dovrebbero essere sempre aperte a estensioni e chiuse a modifiche. Per realizzare questo principio si utilizzano astrazioni come classe astratte o interfacce. Nel caso in cui si voglia modificare un'interfaccia per aggiungere una funzionalità si può creare una nuova classe concreta che implementa questa funzionalità invece di modificare le classi già esistenti.

Per soddisfare il principio si utilizza l'ereditarieta:

- Di interfaccia → Le classi derivare ereditano da una classe base astratta con funzioni virtuali. L'interfaccia è chiusa a modifiche perché non contiene codice
- Di implementazione → Nuove sottoclassi estendono la classe base, mantenendo il codice comune in quest'ultima

Questo approccio consente maggiore modularità del sistema e stabilità, in quanto non si modificano componenti definiti in precedenza e non si introducono errori a causa delle modifiche

▼ Principio di Sostituibilità di Liskov

Il principio afferma che un cliente che usa una classe debba poter utilizzare una sua sottoclasse senza accorgersi del cambiamento.

Questo principio può essere violato a causa della possibilità di ridefinizione dei metodi virtuali. Il principio del **Design By Contract** definisce le precondizioni e postcondizioni:

- **Precondizioni**: requisiti minimi che devono essere soddisfatti dal chiamante per poter utilizzare il metodo
- **Postcondizioni**: requisiti soddisfatti dal metodo in caso di esecuzione corretta

I metodi ridefiniti in una sottoclasse devono avere:

- Precondizioni ugualmente o meno stringenti
- Postcondizioni ugualmente o più stringenti

Inoltre, la semantica della classe base deve essere conservata e non si devono aggiungere vincoli alla classe base.

▼ Principi di Architettura dei Package

Principio di Equivalenza di Riuso/Rilascio (REP)

Un elemento riutilizzabile deve essere gestito da un sistema di rilascio. I client dovrebbero rifiutarsi di utilizzare elementi di cui non si tiene traccia del numero di versione e di cui non si mantengono le versioni precedenti almeno per un po'. In Java l'entità di rilascio è il package, in cui vengono raggruppate le classi.

Principio di Chiusura Comune (CCP)

Maggiore è il numero di package che cambiano in una versione, maggiore è il lavoro di rebuild, test e distribuzione. Occorre ridurre il al minimo il numero di package, in modo che risulti sempre più semplice gestire e modificare ciascuna classe e ridistribuire ciascun package. Perciò, è bene raggruppare classi che vengono modificate insieme nello stesso package.

Principio del Riutilizzo Comune (CRP)

Occorre non raggruppare nello stesso package le classi che non vengono riutilizzate insieme. Questo perché per ciascuna modifica del package, è necessario controllare ogni singola classe per verificare il corretto funzionamento dell'aggiornamento. Una dipendenza da un package, infatti, è una dipendenza da tutto ciò che è al suo interno.

Come si può notare CCP è nettamente in contrasto con CRP e REP, in quanto il primo mira ad avere package che siano più grandi possibile, mentre gli altri due puntano ad ottenere molti package piccoli e specializzati. Una buona prassi è quella di utilizzare CCP in fase di progettazione e poi utilizzare REP e CRP,

facendo il refactoring dei package e permettendo maggiore riutilizzo di questi ultimi.

Esistono anche tre principi che descrivono le relazioni tra i package:

Principio delle Dipendenze Acicliche (ADP)

Le dipendenze tra package non dovrebbero formare dei cicli. Questo perché se si vuole ricompilare una classe si deve ricompilare l'intero package di appartenenza e quelli che dipendono da quest'ultimo. In caso di dipendenze cicliche, una modifica a una classe porterebbe alla ricompilazione di un'intera applicazione.

Per evitare questo problema è sufficiente interrompere un ciclo interponendo un'interfaccia responsabile delle dipendenze cicliche e sfruttando il **principio di inversione delle dipendenze**.

Principio delle Dipendenze Stabili (SDP)

Un package dovrebbe dipendere solo da package più stabili di esso. Un package stabile rispetta il CCP (Principio di Chiusura Comune), dato che le modifiche non causano effetti a cascata sull'intero progetto, ma rimangono interne al package di appartenenza.

Principio delle Astrazioni Stabili (SAP)

I package stabili dovrebbero essere astratti, poiché facili da estendere. Un package astratto non contiene codice soggetto a modifiche.

Bisogna considerare i sistemi di package a livelli: i package dei livelli superiori sono instabili e facilmente modificabili, mentre quelli ai livelli inferiori dell'applicazione sono difficilmente soggetti a modifiche. Una buona prassi è avere package stabili astratti, in modo che possano essere difficili da modificare, ma facili da estendere.

Design Pattern

▼ Design Pattern

I Design Pattern catturano l'esperienza acquisita nella risoluzione di un problema progettuale per poterla riutilizzare in problemi simili. Un Design Pattern è costituito da un nome rilevante, il problema da risolvere, la soluzione al problema e le conseguenze dell'utilizzo del pattern stesso.

Rendono i progetti Object-Oriented più flessibili e riutilizzabili.

Si possono dividere in diverse tipologie:

- **Di creazione** - Risolvono problemi riguardanti la creazione di oggetti
- **Strutturali** - Risolvono problemi riguardanti la composizione di oggetti
- **Comportamentali** - Risolvono problemi riguardanti l'interazione e la distribuzione di responsabilità tra oggetti

▼ Pattern Singleton

Prevede che una classe abbia una sola istanza e che ci sia un punto d'accesso globale ad essa.

Si prevede l'esistenza di un costruttore privato e di un metodo statico per ottenere l'istanza della classe. Se l'istanza esiste già, viene restituita, altrimenti si crea una nuova istanza.

Un'alternativa al pattern Singleton sarebbe l'utilizzo di una classe statica, ma questa non permetterebbe la creazione di istanze personalizzate in base al contesto o l'implementazione di un numero arbitrario di interfacce

▼ Pattern Observer

Viene utilizzato nel caso in cui un oggetto debba notificare un cambiamento del proprio stato interno ad altri oggetti. La relazione tra questi oggetti (subject e observer) potrebbe essere codificata nell'oggetto che cambia (il subject), ma questo dovrebbe conoscere come aggiornare gli osservatori.

La soluzione a questo problema è la definizione di metodi per registrarsi all'interno del Subject e la dichiarazione di un metodo update da parte degli observer. Quando il subject cambia il proprio stato interno, chiama il metodo per notificare gli observer, che eseguiranno il metodo update.

Possibili soluzioni alternative a questo problema sono:

- Class based - ogni subject ha un riferimento agli observer da notificare
- Interface based - il subject possiede un riferimento all'interfaccia implementata dagli observer. In questo modo si aumenta il disaccoppiamento tra subject e observer.
- Delegate based
- Event based

▼ Pattern MVC - MVP

Il model elabora e gestisce lo stato, il controller si occupa dell’input e la view dell’output.

Il model:

- Gestisce i dati
- Risponde a interrogazioni sui dati
- Risponde a istruzioni di modifica dello stato
- Genera eventi al cambiamento dello stato
- Registra gli oggetti interessati alla notifica dell’evento
- In Java estende Observable

Il Controller:

- Gestisce l’input dell’utente
- Mappa le azioni dell’utente in comandi
- Invia i comandi al model e alla view
- In Java è un listener

La View:

- Gestisce la presentazione dei dati all’utente. Mappa i dati del model in oggetti visuali e visualizza gli oggetti su un dispositivo di output
- Si registra presso il model
- In Java estende Observer

▼ Pattern Flyweight

Il pattern Flyweight descrivere come condividere oggetti leggeri tra client diversi così che il loro utilizzo non risulti troppo costoso. Un flyweight è un oggetto condiviso che può essere utilizzato da più client contemporaneamente.

Non deve essere distinguibile da un oggetto non condiviso e non deve fare ipotesi sul contesto in cui opera. I client non istanziano direttamente i flyweight, ma li ottengono tramite una Flyweight Factory.

Si distingue tra:

- Stato intrinseco: non dipende dal contesto di utilizzo ed è memorizzato nel flyweight e condiviso a tutti i client
- Stato estrinseco: dipende dal contesto di utilizzo e non può essere condiviso tra i client. Viene memorizzato o calcolato all'interno del client e passato al flyweight quando viene invocata un'operazione su di esso

Nel diagramma UML troviamo una FlyweightFactory che istanzia e fornisce i diversi Flyweight (che implementano un'interfaccia) al client.

▼ Pattern Strategy

Il pattern Strategy è un pattern comportamentale che viene usato per definire una famiglia di algoritmi intercambiabili tra loro, che implementano una determinata funzionalità. Questi algoritmi sono incapsulati in una gerarchia di classi. Il client dipende solo dall'interfaccia principale per lo svolgimento della funzionalità e non sa quale algoritmo venga effettivamente utilizzato.

(es. algoritmi di ordinamento)

▼ Pattern Adapter

Il pattern Adapter è un pattern di tipo strutturale che permette di convertire l'interfaccia di una classe in quella che il client si aspetta. Fa in modo che interfacce incompatibili tra loro possano lavorare insieme.

Se un client dipende da un'interfaccia già definita che espone un metodo, ed esiste una classe che realizza tale metodo ma la sua interfaccia è incompatibile con quella di Target, si utilizza una classe Adapter che implementa tale interfaccia e chiama il metodo della classe che lo implementa in modo del tutto trasparente per l'utente.

▼ Pattern Decorator

Il pattern Decorator permette di aggiungere dinamicamente una funzionalità a un oggetto. E' un'alternativa flessibile alla specializzazione di una classe.

I vantaggi del pattern Decorator rispetto all'ereditarietà sono i seguenti:

- Consente di aggiungere nuove funzionalità a un oggetto a runtime
- Rende il codice pulito e facile da testare

Il pattern Decorator soddisfa il principio di singola responsabilità, in quanto ogni funzionalità è all'interno di una classe che compie un compito specifico.

Nel diagramma UML si ha una classe Component (Interfaccia o classe astratta) che dichiara l'interfaccia di tutti gli oggetti ai quali si possono aggiungere responsabilità. ConcreteComponent è l'oggetto al quale si possono aggiungere responsabilità e Decorator (classe astratta) mantiene un riferimento a Component e definisce un'interfaccia conforme ad esso. ConcreteDecorator aggiunge responsabilità al component.

▼ Ereditarietà dinamica e Pattern State

Per il principio di sostituibilità di Liskov una classe è sempre più specializzata della sua superclasse. Se si vuole cambiare comportamento di un oggetto in base al suo stato, si potrebbe cambiare la classe dell'oggetto a runtime, ma questo è in possibile.

Una soluzione a questo problema è il pattern State, attraverso il quale un oggetto può comportarsi come se avesse cambiato classe.

Questo pattern localizza il comportamento specifico di uno stato e lo suddivide in funzione di esso. Le classi concrete contengono la logica di transizione da uno stato all'altro. Permette di emulare l'ereditarietà multipla.

▼ Pattern Composite

Il pattern Composite permette di comporre oggetti in una struttura ad albero per rappresentare una gerarchia di contenitore-contenuto e rendere più semplice la manipolazione della struttura.

Attraverso il pattern Composite i client possono trattare allo stesso modo oggetti singoli o composti.

Nel diagramma UML troviamo:

- Component (astratto) - dichiara l'interfaccia e realizza il comportamento di default.
- Client - manipola gli oggetti della composizione attraverso Component
- Leaf - oggetto che non può avere figli
- Composite - oggetto che può avere figli. Il contenitore dei figli è attributo di Composite. Un riferimento esplicito al genitore semplifica l'attraversamento della struttura. L'attributo che contiene il riferimento al genitore e la gestione sono nella classe Component

L'utilizzo del pattern Composite pone diversi obiettivi:

- Massimizzare Component così che il client veda solo lui e utilizzi solo le sue operazioni
- Trasparenza - Il client può utilizzare gli oggetti in modo uniforme. Tuttavia potrebbe effettuare operazioni. I metodi add e remove sono in Component.
- Sicurezza - Le operazioni sui figli (add e remove) vengono messe in Composite e l'invocazione di operazioni illecite sulle foglie genera eccezione. Il client, però, deve gestire due interfacce diverse: bisogna infatti verificare se l'oggetto su cui si vuole agire è un Composite.

▼ Pattern Visitor

Il pattern Visitor permette di definire una nuova operazione da effettuare sugli elementi di una struttura senza dover modificare gli elementi coinvolti.

Il codice di un'operazione viene racchiusa in una singola classe:

- I nodi della gerarchia devono accettare la visita di un Visitor
- Per aggiungere un'operazione basta creare una classe

Il Visitor dichiara un'operazione per ogni tipo di nodo concreto e ogni nodo deve dichiarare un'operazione per accettare un Visitor generico.

Nel diagramma UML:

- Visitor dichiara un metodo visit per ogni classe di elementi concreti
- Concrete Visitor definisce tutti i metodi visit e l'operazione da effettuare e l'operazione da effettuare sulla struttura
- Element dichiara il metodo accept(Visitor)
- Concrete Element definisce il metodo accept(Visitor)
- Structure è la struttura di elementi. Deve poter enumerare i suoi elementi e dichiarare un'interfaccia che permette al client di far visitare la struttura a un visitor

L'utilizzo del pattern Visitor:

- Facilita l'aggiunta di operazioni
- Permette al Visitor di accedere allo stato degli elementi utilizzando l'incapsulamento
- E' difficile aggiungere un elemento concreto alla gerarchia → La gerarchia deve essere stabile

- Ogni Visitor può modificare il proprio stato durante l’operazione
- Il metodo Accept è di tipo double-dispatch perché dipende sia da Element che da Visitor

▼ Pattern Abstract Factory

Il pattern Abstract Factory risolve il problema della creazione di elementi dipendenti senza che il client debba specificare i nomi delle classi concrete nel proprio codice.

Per applicare questo pattern, bisogna soddisfare i seguenti requisiti:

- Il sistema è indipendente dal modo in cui vengono creati gli oggetti
- Il sistema è configurato come scelta tra più prodotti
- I prodotti organizzati in famiglie sono vincolati a essere usati con prodotti della stessa famiglia

Il pattern prevede la presenza di una classe che astrae la creazione di una famiglia di oggetti. Istanze diverse costituiscono implementazioni diverse di membri della stessa famiglia. Le ConcreteFactory creeranno i prodotti.

Conseguenze dell’utilizzo del pattern sono le seguenti:

- Isola le classi concrete
- Si può cambiare facilmente la famiglia di prodotti utilizzata
- Coerenza nell’utilizzo dei prodotti
- E’ difficile aggiungere supporto per nuove tipologie di prodotti

Sistemi di Controllo delle Versioni

▼ Modello Lock-Modify-Unlock (LMU)

E’ un modello utilizzato nei sistemi di controllo delle versioni. Un utente che vuole modificare un file all’interno del repository (posto in cui vengono salvati i file correnti e la cronologia) deve bloccare il file e sbloccarlo una volta terminate le modifiche. Un file può essere modificato da una sola persona alla volta.

Questo modello introduce numerosi svantaggi:

- Ci si può dimenticare di sbloccare il file

- Serializzazione non necessarie: due modifiche in punti diversi del file non portano a conflitti, ma comunque non è possibile effettuarle contemporaneamente
- Esiste ancora la possibilità di incompatibilità delle modifiche: se vengono modificati due file che dipendono l'uno dall'altro, questi due potrebbero non funzionare dopo le modifiche
- Non si può lavorare offline

LMU è comodo solo se i file non sono unibili.

▼ Modello Copy-Modify-Merge (CMM)

In questo modello un utente copia il contenuto del repository in una copia di lavoro (copia locale su cui viene effettuato il lavoro. Agisce come una sandbox, in quanto le modifiche non influiscono sul repository principale), effettua le modifiche e poi effettua il Check in delle modifiche (commit). Una vota effettuato il check in, si fa il merge delle modifiche con il contenuto del repository. Possono presentarsi due situazioni diverse:

- Successo
- Conflitto - lo stesso blocco di codice è stato modificato in contemporanea da un altro utente. I conflitti vanno risolti manualmente.

L'assenza di conflitti non implica che le modifiche funzionino correttamente.

Rispetto a LMU, CMM apporta numerosi vantaggi:

- E' più fluido
- Si può lavorare in parallelo
- Di solito le modifiche non si sovrappongono, quindi si hanno pochi conflitti
- I conflitti richiedono poco tempo per essere risolti

▼ Distributed Version Control System (DVCS)

Il tuo riassunto sui Distributed Version Control Systems (DVCS) è abbastanza accurato, ma ci sono alcune inesattezze e concetti che possono essere approfonditi ulteriormente. Ecco una revisione del tuo testo:

Un Distributed Version Control System (DVCS) permette la sincronizzazione di set di modifiche tra peer, senza la necessità di un server centrale. Ogni peer ha una copia di lavoro completa del repository, che include il codice e la cronologia

delle modifiche. Le operazioni comuni sono veloci perché non richiedono comunicazioni con un server centrale.

Le revisioni in un DVCS sono organizzate come una linea di sviluppo con la possibilità di creare branch. Questo spesso porta a una struttura di grafo diretto aciclico, in cui le revisioni si collegano tra loro in una serie di commit.

I concetti principali di un DVCS sono:

- **Trunk**: La linea principale delle revisioni, che rappresenta lo sviluppo principale del progetto.
- **Branch**: Una diramazione da una specifica versione del repository. I branch consentono lo sviluppo indipendente di nuove funzionalità o correzioni di bug. Successivamente, possono essere riuniti (merged) con il trunk o con altri branch.
- **Tag**: Un'etichetta che fa riferimento a una specifica istanza nel tempo del repository. I tag sono spesso utilizzati per marcare versioni stabili o importanti del codice.
- **Push/Pull**: Operazioni per trasferire le modifiche tra repository, sia da una copia locale a un repository remoto (push), sia da un repository remoto a una copia locale (pull).

Se si utilizza un DVCS, i passi tipici sono i seguenti:

1. **Commit**: Spostamento delle modifiche dalla copia di lavoro al repository locale, creando una nuova revisione.
2. **Push**: Trasferimento delle modifiche dal repository locale a un repository remoto, rendendole disponibili ad altri utenti o peer.
3. **Pull**: Trasferimento delle modifiche da un repository remoto alla copia locale, permettendo di ottenere le modifiche apportate da altri utenti.
4. **Update**: Aggiornamento della copia locale con le nuove modifiche prese dal repository.

I DVCS offrono numerosi vantaggi, tra cui:

- **Sandbox locale**: Ogni utente ha una propria copia di lavoro completa del repository, in cui può sperimentare e apportare modifiche senza interferire con il lavoro degli altri.
- **Modalità offline**: Un DVCS permette di lavorare offline, senza la necessità di una connessione continua al server centrale.

- **Velocità**: Le operazioni comuni, come il commit e l'aggiornamento, sono veloci perché avvengono localmente, senza la necessità di comunicazioni remote.
- **Branching e merging semplici**: I DVCS semplificano la creazione di nuovi branch e la fusione delle modifiche tra di essi, agevolando lo sviluppo collaborativo e il controllo delle versioni.
- **Gestione ridotta**: Con un DVCS, la necessità di gestire manualmente i file e le versioni diminuisce grazie alla natura distribuita del sistema.

Tuttavia, hanno anche alcuni svantaggi svantaggi:

- **Backup**: Anche se un DVCS offre una copia di lavoro locale come backup, è comunque importante eseguire regolari backup del repository per proteggere le modifiche e la cronologia delle revisioni da perdite accidentali.
- **Ultima versione**: In un DVCS, non esiste un'unica "ultima versione" del codice. Invece, ci sono diverse revisioni che rappresentano lo stato del repository in tempi diversi.
- **Numeri di versione e tag**: Anche se i DVCS non forniscono numeri di versione automatici, è possibile utilizzare i tag per contrassegnare punti importanti o significativi nella cronologia del repository.

Complessivamente, i DVCS offrono un modo flessibile e distribuito per gestire il controllo delle versioni, consentendo una maggiore collaborazione tra gli sviluppatori e una gestione più efficiente del codice sorgente.

▼ Git

Git è un popolare sistema di controllo delle versioni distribuito (DVCS) che utilizza i seguenti concetti:

- **Repository**: Una directory che contiene la cronologia delle modifiche di un progetto. È la base di archiviazione di tutte le versioni del codice sorgente e delle informazioni di revisione.
- **Directory di lavoro**: Una copia locale del repository che consente agli sviluppatori di lavorare sui file del progetto. La directory di lavoro riflette uno stato specifico del repository, come definito da una revisione o un branch.
- **Index**: Anche chiamato "Area di staging" o "Area di sosta", è un'area intermedia in cui vengono elencate le modifiche apportate ai file della directory di lavoro che verranno incluse nel prossimo commit. L'index

consente di selezionare in modo selettivo le modifiche da aggiungere al repository.

- **Commit**: Il processo di salvataggio delle modifiche apportate ai file nella repository. Un commit rappresenta una singola revisione nel tempo e include una descrizione delle modifiche effettuate.

Le corrette fasi da seguire utilizzando Git sono le seguenti:

1. **Clone**: Clonare un repository esistente per creare una copia locale nella propria directory di lavoro.
2. **Modifica dei file**: Apportare le modifiche necessarie ai file nella directory di lavoro. Queste modifiche possono includere l'aggiunta, la modifica o l'eliminazione di file.
3. **Aggiunta all'Index**: Selezionare le modifiche specifiche che si desidera includere nel commit, aggiungendo i file modificati all'index. Questo può essere fatto con il comando `git add`.
4. **Commit**: Salvare le modifiche selezionate dall'index nella repository come una nuova revisione. Ogni commit ha un messaggio che descrive le modifiche apportate.
5. **Push**: Trasferire le nuove revisioni dal repository locale a un repository remoto, rendendole disponibili ad altri utenti o peer. Questo può essere fatto con il comando `git push`.

Alcuni concetti aggiuntivi da considerare quando si lavora con Git includono:

- **Branch**: Un ramo indipendente della linea di sviluppo che consente di lavorare su nuove funzionalità o correzioni di bug senza influire sulla linea principale del progetto (trunk). I branch consentono di sviluppare in modo isolato e possono essere uniti (merged) con altri branch o con il trunk quando le modifiche sono pronte.
- **Status**: Utilizzando il comando `git status`, è possibile visualizzare lo stato attuale del repository, inclusi i file modificati, i file in sosta, i commit effettuati e altre informazioni utili.
- **Pull**: Aggiornare la propria copia locale del repository con le nuove revisioni prese dal repository remoto. Questo può essere fatto con il comando `git pull`.

Complessivamente, Git fornisce una solida infrastruttura per la gestione delle versioni dei progetti, permettendo agli sviluppatori di lavorare in modo collaborativo, tenere traccia delle modifiche e coordinare il flusso di lavoro in modo efficace.

Modelli e Processi di Sviluppo

▼ Modelli e modelli di processo

Un modello è un insieme di concetti e proprietà per catturare aspetti essenziali di un sistema. Rappresenta una versione semplificata della realtà che cattura per la facile accessibilità alla comprensione e valutazione.

I processi di sviluppo basati su modelli rendono esplicite le conoscenze sul sistema attraverso la costruzione di diagrammi formali.

Un concetto fondamentale per i modelli di processo è quello di **tracciabilità**. Per tracciabilità si intende che in qualsiasi verso si percorra la sequenza dei modelli, si devono poter mappare uno o più elementi del modello in uno o più elementi di un altro. La tracciabilità serve a:

- Garantire coerenza e consistenza tra modelli
- Creare un processo logico tra requisiti e codice
- Tenere le modifiche sotto controllo

I **linguaggi di modellazione** sono utilizzati per descrivere un sistema (modello software). Lo stesso modello può essere rappresentato da linguaggi differenti con diverso potere espressivo.

Questi linguaggi devono soddisfare alcuni requisiti, tra cui:

- Essere abbastanza precisi
- Essere flessibili dal punto di vista descrittivo
- Possibilmente essere standard

Il codice, ad esempio, è il più dettagliato, ma fornisce una visione piatta del modello e non evidenzia i punti salienti. Non si ha una visione d’insieme a primo impatto. Per mantenere la tracciabilità, una buona pratica è quella di modificare il modello di progettazione e poi generare automaticamente il codice da esso quando è possibile.

Il linguaggio di UML più completo e che soddisfa tutti i requisiti sopra elencati è **UML** (Unified Modeling Language).

Un **processo di sviluppo**, infine, è un insieme di passi ordinati che coinvolge attività, vincoli e risorse per produrre il risultato desiderato.
Ogni fase è una porzione di lavoro svolto che rispetta un insieme di politiche, strutture organizzative, tecnologie, procedure, …, per concepire, sviluppare, distribuire e manutenere il prodotto software.
Le fasi generiche di un modello di processo sono: specifica, sviluppo, validazione ed evoluzione.
Un modello di processo è una versione semplificata di un processo da uno specifico punto di vista.

▼ Modelli a Cascata

Il modello a cascata identifica una serie di fasi distinte in cascata tra loro. Prende vita dal presupposto che introdurre cambiamenti nelle fasi più avanzate ha costi troppo elevati, perciò ogni fase dovrebbe essere svolta nel modo più esaustivo possibile per evitare di generare retroazioni.

Nel modello a cascata bisogna definire:

- Semilavorati: documentazione redatta in una determinata fase e utilizzata da quella successiva, che ne garantisce la validità
- Date di scadenza entro cui completare il lavoro svolto per una fase

Il modello a cascata ha alcuni limiti derivanti dal fatto che si basa su due presupposti non realistici:

- Immutabilità dell’analisi
- Immutabilità del progetto

Questi due vincoli non sono quasi mai realizzati nella realtà, in quanto spesso i clienti modificano i propri requisiti durante le fasi dello sviluppo del sistema e spesso l’implementazione modifica la progettazione perché le idee migliori vengono in mente in questa fase. Ciò rende il modello a cascata particolarmente rigido, seppure questo approccio porti ad avere un maggiore controllo dell’andamento del progetto.

Una soluzione immediata potrebbe essere quella di avere le diverse fasi in retroazione con quella precedente, o anche la creazione di un prototipo che mostra al cliente le funzionalità che avrà il sistema e che verrà poi abbandonato

durante la progettazione. Tuttavia, la creazione di un prototipo è un processo dispendioso che annulla i vantaggi portati dall'economicità del modello a cascata.

▼ Modelli Evolutivi - Extreme Programming

I modelli evolutivi partono da specifiche astratte creando un prototipo che viene man mano raffinato. Si lavora a stretto contatto con il cliente.

L'esasperazione di questo tipo di modelli è l'Extreme Programming, in cui si modifica il codice dell'applicazione ogni volta che bisogna fare una modifica, senza passare per il documento di progettazione. Questo tipo di modello implica la comunicazione tra sviluppatori e con il cliente, un corposo lavoro di testing per assicurare che il sistema funzioni correttamente e una particolare abilità nella programmazione. Si punta sempre alla semplicità, non pensando al riutilizzo futuro. L'Extreme Programming ha numerosi problemi, tra cui:

- Il processo di sviluppo non è visibili
- Il sistema è poco strutturato
- E' richiesta una grande capacità nella programmazione

▼ Modelli Ibridi, Sviluppo Incrementale, Sviluppo Iterativo

I modelli ibridi sono composti da sottosistemi. Per ogni sottosistema è possibile adottare un modello di sviluppo divevrso.

Sviluppo Incrementale

Prevede la costruzione del sistema sviluppando in sequenza parti ben definite e non più modificabili. Si specificano perfettamente tutti i requisiti prima di implementarli.

Sviluppo Iterativo

Si effettuano molti passi dell'intero ciclo di sviluppo del software e si costruisce ogni volta il sistema aumentando il livello di dettaglio dei componenti. Non funziona bene per progetti di grandi dimensioni.

Sviluppo Incrementale-Iterativo

Si individuano sottoparti autonome del sistema e si realizza il prototipo di una di esse. Si continua il procedimento con le altre parti del sistema e si aumenta

sempre più l’estensione e il dettaglio dei prototipi tenendo in considerazione tutte le parti interagenti.

▼ Rational Unified Process - RUP

RUP è un modello di processo iterativo ibrido che contiene elementi di tutti i modelli di processo generici. Definisce un framework adattabile che può dar luogo a diversi processi in contesti diversi.

In RUP esistono tre diverse visioni del processo di sviluppo:

- Prospettiva dinamica delle fasi nel tempo
- Prospettiva statica delle attività coinvolte
- Prospettiva pratica che definisce le buone prassi da seguire

Prospettiva Dinamica

E’ composta da più fasi:

1. Avvio - delinea il business case, ovvero il tipo di mercato a cui è rivolto il prodotto software
    - Identifica tutte le entità esterne e definisce le interazioni con esse
    - Si utilizza il modello dei casi d’uso, la pianificazione iniziale, la valutazione dei rischi e una prima definizione dei requisiti
2. Elaborazione - fase in cui si definisce la struttura del problema
    - Analisi del dominio e prima progettazione dell’architettura
    - Bisogna soddisfare alcuni criteri:
        - Modello dei casi d’uso quasi completo
        - Descrizione dell’architettura
        - Sviluppo di un’architettura che dimostri il completamento dei casi d’uso significativi
        - Revisione del business case e dei rischi
        - Pianificazione del progetto complessivo
3. Costruzione - si progetta e programma il sistema. Lo sviluppo delle parti del sistema avviene in parallelo e poi si integrano le diverse parti. Si collauda tutto il sistema in modo da ottenere un prodotto finito e funzionante

4. Transizione - rilascio del prodotto software e verifica del rispetto dei requisiti tramite beta testing. Si istruiscono gli utenti all’utilizzo del software

Prospettiva Statica

Ci si focalizza sulle attività necessarie per lo sviluppo di un software:

- Modellazione delle attività aziendali
- Requisiti
- Analisi e progettazione
- Implementazione
- Test
- Rilascio

Queste fasi sono accompagnate da dei workflow di supporto:

- Gestione della configurazione e delle modifiche
- Gestione del progetto
- Ambiente

Prospettiva Pratica

Suggerisce delle buone prassi da seguire nello sviluppo di un sistema:

- Sviluppo ciclico del software partendo dalle funzionalità con priorità più alta
- Gestione dei requisiti (analisi della fattibilità dell’introduzione di nuovi requisiti e decisione sull’introduzione o meno di questi ultimi nel sistema)
- Usare architetture basate sui componenti
- Creare modelli visivi del software
- Verificare la qualità del software
- Controllare le modifiche al software

▼ Tipologie di Requisiti

I requisiti rappresentano la descrizione dei servizi che vengono forniti dal sistema e dei suoi vincoli. Sono divisi in:

- **Requisiti utente**: servizi offerti dal sistema e vincoli su cui opera. Sono molto astratti e di alto livello

- **Requisiti di sistema**: specificano le funzioni, i servizi e i vincoli del sistema in modo dettagliato
    - **Requisiti funzionali**: elencano i servizi che offre il sistema, in particolare per ciascun servizio specificano come il sistema dovrebbe reagire all'input, come comportarsi in determinate situazioni e cosa il sistema non dovrebbe fare
    - **Requisiti non funzionali**: descrivono le caratteristiche del sistema, tra cui:
        - Requisiti del prodotto: limitano le proprietà del sistema
        - Requisiti organizzativi: limitano il processo di sviluppo
        - Requisiti esterni: derivano da sistemi esterni o da contesti come la legislazione sulla privacy o requisiti etici
    - **Requisiti di dominio**: derivano dal dominio del sistema e indicano il funzionamento del sistema in uno specifico dominio.

Sicurezza

▼ GDPR - General Data Protection Regulation

GDPR è la nuova normativa per la realizzazione di sistemi software, che devono rispettare determinati requisiti di protezione della privacy e sicurezza:

- Privacy by design and by default
- Minimalità e proporzionalità
- Anonimizzazione e pseudonimizzazione
- Attenzione al trasferimento dei dati al di fuori dell'Unione Europea
- Adeguatezza delle misure di sicurezza.

In accordo con GDPR, i dati raccolti devono essere:

- trattati in modo lecito, equo e trasparente
- raccolti per finalità definite e legittime
- adeguati, pertinenti e limitati alle finalità per cui vengono raccolti
- esatti e aggiornati
- conservati solo per il tempo necessario a consentire l'identificazione

▼ Sicurezza, Crittografia e Biometria

L’obiettivo di un attacco è il contenuto informativo di un sistema. Bisogna cercare di impedire l’accesso a utenti non autorizzati e regolamentare l’accesso in modo da evitare il furto o la manomissione dei dati.

Bisogna proteggere la riservatezza, integrità, autenticità e disponibilità dell’informazione. Inoltre, è necessario prevenire attacchi DoS che impediscono la disponibilità delle funzionalità del sistema.

Un concetto fondamentale per garantire la sicurezza è quelli di **crittografia**. Essa può avvenire mediante chiave simmetrica o asimmetrica.

- **Chiave simmetrica**
  Garantisce la riservatezza, ma non può né identificare né autenticare. Viene implementata attraverso dispositivi, algoritmi e chiavi segrete. Si utilizza una singola chiave sia per cifrare le informazioni che per decifrarle.
- **Chiave asimmetrica**
  Garantisce la riservatezza e può essere utilizzata per identificare e autenticare. Il concetto di chiave asimmetrica si basa sulla teoria della complessità computazionale. Vengono utilizzate due chiavi correlate ma non facilmente calcolabili (chiave pubblica e chiave privata). La chiave privata è personale e può essere verificata da quella pubblica corrispondente

Un altro concetto fondamentale è quello di **biometria**, che viene spesso utilizzata per l’identificazione ma sarebbe più efficiente se utilizzata per l’autenticazione, in cui c’è bisogno di effettuare un solo confronto con meno probabilità di errore. Ha prestazioni leggermente inferiori rispetto ad altre tecniche ed il suo recupero è impossibile se viene compromessa.

▼ Politica di Sicurezza

Una politica di sicurezza tiene conto dei vincoli tecnici, logistici, amministrativi e politici dell’organizzazione in cui opera il sistema. E’ necessario introdurre la sicurezza sin dalle prime fasi dell’analisi dei requisiti di un nuovo sistema.

E’ necessario, durante la progettazione di un nuovo prodotto software, tenere conto dell’esistenza di sistemi critici, cioè sistemi da cui dipendono persone o aziende. Per i sistemi critici viene fatta una classificazione in base alle conseguenze del loro fallimento:

- **Safety-Critical**: I fallimenti possono provocare incidenti, perdita di vita umane o danni ambientali

- **Mission-Critical**: I fallimenti provocano il fallimento di intere attività o obiettivi
- **Business-Critical**: I fallimenti portano a costi elevati

La proprietà più importante di un sistema critico è la sua **fidatezza**, cioè un insieme di disponibilità, affidabilità, sicurezza e protezione. I sistemi non affidabili, sicuri o protetti sono rifiutati dagli utenti, i costi di un fallimento del sistema possono essere enormi e i sistemi inaffidabili possono provocare perdita di informazioni.

I componenti del sistema che possono provocare fallimenti sono: l'hardware, il software e la componente umana.

La sicurezza dei sistemi è messa a rischio soprattutto dalla sua esposizione sulla rete, ma questo fattore è utile anche per la divulgazione e facile correzione delle vulnerabilità.

Infine, per rendere un sistema sicuro è bene conoscere quali sono gli attacchi che possono colpirlo, ad esempio: exploit, buffer overflow, shell code, sniffing, cracking, spoofing, trojan, DoS.

Per la corretta protezione di un sistema è necessario sia progettare il software in modo sicuro che rendere sicura l'infrastruttura (problema manageriale) attraverso una corretta configurazione.

Le minacce per un sistema possono essere classificate in:

- Minacce per la riservatezza di sistemi e dati
- Minacce all'integrità
- Minacce alla disponibilità

e sono associate a diverse categorie di controlli:

- Per garantire che gli attacchi non abbiano successo
- Per identificare e respingere gli attacchi
- Per il ripristino

▼ Valutazione del Rischio e Analisi dei Beni

L'analisi del rischio consiste nella valutazione delle possibili perdite del sistema in seguito a un attacco e bilanciare le perdite con i costi che esse causerebbero. Occorre effettuare un'attenta valutazione delle politiche di sicurezza dell'organizzazione, che si compone di:

- Valutazione preliminare del rischio, che determina i requisiti di sicurezza dell'intero sistema
- Ciclo di vita di valutazione del rischio, che avviene durante tutto il ciclo di vita dello sviluppo del software.
  In questa fase occorre conoscere l'architettura del sistema, l'organizzazione dei dati, la scelta della piattaforma e del middleware. Vengono effettuate l'individuazione e la valutazione delle vulnerabilità

La valutazione del rischio porta a un insieme di decisioni ingegneristiche che influenzano la progettazione o l'implementazione del sistema e limitano il suo utilizzo.

L'analisi dei beni consiste nell'analisi delle risorse fisiche e logiche e delle relazioni tra esse:

- **Risorse Fisiche**: il sistema è visto come un insieme di dispositivi
  - Individuazione delle risorse
  - Ispezione e valutazione dei locali che le ospitano
  - Verifica della cablatura dei locali
- **Risorse Logiche**: il sistema è visto come insieme di informazioni, flussi e processi
  - Classificazione in base al valore, grado di riservatezza e contesto delle risorse
  - Classificazione dei servizi in modo che non abbiano effetti collaterali
- **Dipendenza tra le risorse**
  - Evidenziare le risorse critiche da cui dipendono molte altre

▼ Identificazione delle Minacce e Valutazioni

L'identificazione delle minacce consiste con la definizione di tutto ciò che non deve accadere nel sistema, considerando come eventi indesiderati tutti gli accessi non permessi.

Gli attacchi intenzionali sono caratterizzati in funzione della risorsa attaccata e delle tecniche usate. Possiamo classificare gli attacchi in:

- **Attacchi a livello fisico**
  - Furto o danneggiamento → Guasti a dispositivi del sistema o di supporto

- **Attacchi a livello logico**
  - Sottrarre informazioni o degradare il sistema
  - Possono implicare la perdita di password o chiavi, la cancellazione di dati e la corruzione del software

Occorre fare un'attenta valutazione, associando un rischio ad ogni minaccia. Un rischio, per definizione, è composto dalla probabilità che l'evento accada per il danno che questo porterebbe. Bisogna valutare attentamente la probabilità che un attacco venga effettuato, anche in base alla facilità di attuazione di quest'ultimo. Infatti, è bene ricordare che un attacco dipende anche dalle risorse che l'attaccante ha a disposizione, perciò è bene tenere in considerazione anche la possibilità di un attacco composto da molti attacchi semplici sequenziali.

Per neutralizzare gli attacchi occorre effettuare:

- **Valutazione del rapporto costo/efficacia**
  - Per valutare il grado di adeguatezza del controllo bisogna tenere in considerazione:
    - Costo della messa in opera del controllo
    - Peggioramento dell'ergonomia dell'interfaccia utente
    - Decadimento delle prestazioni
- **Analisi di standard e modelli di riferimento**
- **Controlli di carattere organizzativo**
  - Affinché la protezione del sistema sia efficace deve essere eseguita nel modo corretto da personale esperto
  - Bisogna definire norme comportamentali per ogni ruolo nell'organizzazione
- **Controlli di carattere tecnico**
  - Controllo sui diritti di accesso
  - Moduli software di cifratura
  - Apparecchiature di telecomunicazioni che cifrano il traffico in modo trasparente alle applicazioni
  - Firewall e Server Proxy

    - Chiavi e dispositivi di riconoscimento biofisici

Occorre, quindi, effettuare un insieme di controlli adottando un sottoinsieme di costo minimo che rispetti alcuni vincoli, cioè completezza, omogeneità, ridondanza controllata ed effettiva attuabilità.

▼ Misuse Case, Security Use Case e Requisiti di Sicurezza

I Misuse Case riguardano l'interazione tra l'applicazione e l'attaccante. Sono adatti per analizzare le minacce ma non dicono nulla riguardo i modi per evitarle. A questo proposito si utilizzano i Security Use Case, che specificano i requisiti tramite i quali l'applicazione si protegge dagli attacchi.

Lo scopo dei requisiti di sicurezza è di definire quali comportamenti risultano inaccettabili per il sistema, senza definire funzionalità richieste al sistema. Specificano il contesto dei beni da proteggere e il valore che questi hanno per l'organizzazione.

- Requisiti di identificazione
- Requisiti di autenticazione
- Requisiti di autorizzazione
- Requisiti di immunità - meccanismi di difesa da minacce
- Requisiti di integrità - come evitare la corruzione di dati
- Requisiti di scoperta delle intrusioni - meccanismi per la rilevazione degli attacchi
- Requisiti di non ripudiazione
- Requisiti di riservatezza - come mantenere riservate le informazioni
- Requisiti di controllo della protezione - come controllare e verificare il corretto utilizzo del sistema
- Requisiti di protezione della manutenzione del sistema - come evitare modifiche non autorizzate nel caso in cui vengano annullati i meccanismi di protezione

▼ Progettazione per la Sicurezza

Occorre tenere conto della sicurezza del sistema sin dalle prime fasi della progettazione, perché non è possibile ottenere un'implementazione sicura da una

progettazione non sicura.

L’architettura del sistema, definita in fase di progettazione, gioca un ruolo fondamentale per la sicurezza, in quanto è necessario tenere conto di:

- **Protezione**: come organizzare i dati nel sistema in modo che siano protetti
- **Distribuzione**: come distribuire i beni per minimizzare gli effetti di un attacco

I due problemi sono potenzialmente in conflitto, in quanto tenere i dati nello stesso posto li rende più facili da proteggere, invece la distribuzione di questi aumenta il costo di protezione, ma limita li effetti di un attacco.

La migliore architettura è quella a **Layer** (strati), in cui i beni critici da proteggere sono nei livelli più bassi. In questa architettura, le credenziali di accesso a servizi in strati diversi dovrebbero essere diversificate.

L’architettura **Client/Server**, invece, è utile se la protezione è un requisito critico, in quanto i dati si trovano sul server, su cui andranno adottati meccanismi di protezione. L’architettura Client/Server classica, tuttavia, presenta delle limitazioni, in quanto se la sicurezza viene compromessa si hanno perdite più ampie e più alti costi di recupero. Inoltre, la presenza di un solo server rende il sistema più soggetto ad attacchi DoS. La soluzione è adottare un’architettura Client/Server distribuita.

Linee guida di progettazione per la sicurezza

Esistono delle linee guida su come progettare un sistema, che servono come mezzo per migliorare la consapevolezza dei problemi di sicurezza e forniscono una base per una lista di controlli da effettuare sul sistema.

- Avere un’esplicita politica di sicurezza
- Evitare un singolo punto di fallimento
    - Se non si ha un singolo punto di fallimento, un fallimento di una parte del sistema non fa fallire tutto il sistema.
    - Non si dovrebbe utilizzare un singolo meccanismo per la sicurezza, ma tecniche differenti.
- Fallire in modo sicuro
    - Si dovrebbe fallire in modo sicuro, con procedure di fall-back sicure almeno quanto il sistema. Anche se il sistema fallisce l’attaccante non dovrebbe poter accedere a dati riservati del sistema

- Bilanciare sicurezza e usabilità
- Consapevolezza dell’ingegneria sociale
- Usare ridondanza e diversità
    - Ridondanza: avere più versioni del software e dei dati
    - Diversità: utilizzare piattaforme e tecnologie diverse per le versioni diverse, così che le vulnerabilità di una singola piattaforma o tecnologia non influenzino tutte le versioni
- Compartimentizzazione dei beni
    - Organizzare le informazioni del sistema in modo che gli utenti abbiano accesso solo a quelle necessarie
- Progettare per il deployment e per il ripristino
    - Il deployment coinvolge la configurazione del sistema e dell’ambiente, l’installazione del sistema, la configurazione del sistema installato. In questa fase spesso si introducono molte vulnerabilità. Buone pratiche sono:
        - Includere un supporto per la visualizzazione delle configurazioni
        - Minimizzazione dei privilegi di default
        - Localizzazione delle impostazioni di configurazione
        - Fornire modi per rimediare alle vulnerabilità
        - Inclusione di meccanismi per l’aggiornamento della versione

La Security Policy è un documento che definisce la sicurezza, ma non definisce come implementarla. Le politiche di sicurezza vengono inserite nella progettazione per:

- Specificare come accedere alle informazioni
- Specificare quali informazioni testare per l’accesso
- Specificare a chi concedere l’accesso

Esiste un insieme di regole e condizioni incorporate nella Security Authority del sistema. Sono divise in categorie:

- **Identity Policies -** regole per la verifica delle credenziali
- **Access Control Policies -** regole per l’accesso a risorse e funzionalità

- **Content-Specific Policies -** regole per informazioni specifiche durante la memorizzazione e comunicazione
- **Network and Infrastructures Policies -** regole per controllare il fllusso di dati e il deployment su reti e infrastrutture
- **Regulatory Policies -** regole per l'applicazione e per il rispetto dei requisiti legali
- **Advisor and Information Policies -** regole dell'organizzazione

▼ White Box Testing e Black Box Testing

- Black Box Testing
  - Si assume di non conoscere com'è fatta l'applicazione e non si ha a disposizione il codice sorgente. Si eseguono i test sulle vulnerabilità fingendo di essere dei veri attaccanti. Si utilizzano tool per l'analisi e i tentativi di sfruttamento delle vulnerabilità
  - Si attacca l'infrastruttura per scovare errori di configurazione di host e reti, falle di sicurezza delle macchine virtuali e problemi dei linguaggi di programmazione
- White Box Testing
  - Si ha la completa conoscenza dell'applicazione e si ha a disposizione il codice sorgente. Si fanno delle revisioni del codice per trovare debolezze e vulnerabilità, anche attraverso tool di debugging.
  - Vengono scoperti problemi tipici come corse critiche, mancata validazione dell'input, memory leak e problemi di stabilità o prestazioni.

▼ Capacità di Sopravvivenza del Sistema

Per capacità di sopravvivenza si intende la capacità di un sistema di continuare a fornire servizi mentre è sotto attacco o dopo che un attacco ha avuto successo e ha danneggiato parti di esso.

Per garantirla, bisogna conoscere:

- quali sono i servizi critici
- come possono essere compromessi
- la qualità minima dei servizi che deve essere mantenuta
- come proteggere i servizi critici

- come ripristinare velocemente il sistema

Il Survival Analysis System è un metodo ideato per garantire la capacità di sopravvivenza di un sistema. Serve a valutare le vulnerabilità del sistema e a supportare la progettazione di architetture che possono sopravvivere ad attacchi.

Si compone di strategie:

- **Resistenza** - si costruisce nel sistema la capacità di resistere ad attacchi
- **Identificazione** - si costruisce nel sistema la capacità di riconoscere attacchi e fallimenti e valutare il danno
- **Ripristino** - si costruisce nel sistema la capacità del sistema di fornire servizi essenziali durante un attacco, per poi ripristinarli tutti

e fasi:

- **Comprensione del sistema**
- **Identificazione dei servizi critici**
- **Simulazione degli attacchi**
- **Analisi della sopravvivenza**